x86-64 Control Flow
Topics

- Control flow
- Condition codes
- Jumps and conditional moves
- “Goto code”
- Loops
- Translating from C to x86-64
Motivation

- We cannot translate the following C function to assembly, using only data movement and arithmetic operations
  - Fundamental requirement: ability to control the flow of program execution (i.e., decision-making)
  - Necessary for translating structured code

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```
Control flow

- The **program counter (PC)** tracks the address of the next instruction to be executed
  - To change the PC in assembly, use a **jump** instruction
    - Often the jump will be relative to the current PC value
  - In assembly, the target of a jump is usually a **label**, which is converted to a code address by the assembler
    - Labels are written using colon notation (e.g., “L1:`")
  - However, **unconditional** jumps aren’t sufficient for decision-making
    - In fact, the compiler can just re-arrange code to avoid them

```
movl $2, %eax
jmp L1
movl $3, %eax        # never executed!
L1:
movl $4, %eax
```
Conditional jumps

- Conditional jumps only jump under certain conditions
- In machine/assembly code, conditional jumps are often encoded using a pair of instructions
  - The first sets the condition codes of the CPU
    - On x86-64, the FLAGS register
    - Arithmetic/logical instructions do this as a side effect
    - Special-purpose instructions `cmp` and `test`
  - The second jumps base on the value of the condition codes
    - On x86-64, many variants: “jump-if-equal”, “jump-if-less-than”, etc.

```asm
  cmpl %eax, %ecx           # means “compare %ecx with %eax”
jle pos1                   # means “jump-if-less-than-or-equal”
```
Condition codes

- x86-64: special %flags register stores bits for these condition codes:
  - **CF** (carry): last operation resulted in a carry out or **borrow in**
    - (e.g., overflow for unsigned arithmetic)
  - **ZF** (zero): last operation resulted in a zero
  - **SF** (sign): last operation resulted in a negative value
  - **OF** (overflow): last operation caused a two's complement overflow (negative or positive)

- As well as a few we won’t use:
  - **PF** (parity): last operation resulted in an even number of 1 bits in the eight least significant bits
  - **AF** (adjust): last operation resulted in a carry out for the four least significant bits
  - **IF** (interrupt): CPU will handle interrupts

- Use $eflags to reference this register in GDB
  - E.g., “print $eflags” or “display $eflags”
Condition codes

- **In addition**, the carry flag is set if an addition requires a carry out of the most significant (leftmost) bit
  - Basically, it’s the “extra bit” necessary to represent the result
  - E.g., \(1001 + 0001 = 1010\) (CF=0)
  - E.g., \(1111 + 0001 = 0000\) (CF=1)

- **In subtraction**, the carry (borrow) flag is set if a subtraction requires a borrow into the most significant (leftmost) bit
  - E.g., \(1000 - 0001 = 0111\) (CF=0)
  - E.g., \(0000 - 0001 = 1111\) (CF=1)
Condition codes

- Special **cmp** and **test** instructions
  - **cmp** compares two values (computes $arg_2 - arg_1$)
    - **NOTE REVERSED ORDERING** – also, the result is not saved
    - Type-agnostic: all flags are set, but not all are relevant!
    - Does not change either operand
  - **test** checks for non-zero values (computes $arg_2 \& arg_1$)
    - Often, the arguments are the same (or one is a bit mask)
    - Always sets carry and overflow flags to zero
    - Does not change either operand

```assembly
cmpl %eax, %ecx       # means “compare %ecx with %eax”

testl $0xFF, %edx     # means “check low-order 8 bits of %edx”
```
Question

- Suppose %rax = 5 and %rbx = 10. Which flag(s) will be set after the following instruction?
  
  cmpq %rax, %rbx  # computes %rbx - %rax

  - A) CF
  - B) ZF
  - C) SF
  - D) None
Question

• Suppose %rax = 5 and %rbx = 10. Which flag(s) will be set after the following instruction?
  
cmpq %rbx, %rax # computes %rax - %rbx

  - A) CF
  - B) ZF
  - C) SF
  - D) None
Suppose $\%rax = 5$ and $\%rbx = 10$. Which flag(s) will be set after the following instruction?

testq $\%rax$, $\%rbx$  # computes $\%rbx \& \%rax$

- A) CF
- B) ZF
- C) SF
- D) None
# Jump instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Jump condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Label</td>
<td>1</td>
<td>Direct jump</td>
</tr>
<tr>
<td>jmp</td>
<td>*Operand</td>
<td>1</td>
<td>Indirect jump</td>
</tr>
<tr>
<td>je</td>
<td>Label</td>
<td>jz</td>
<td>ZF</td>
</tr>
<tr>
<td>jne</td>
<td>Label</td>
<td>jnz</td>
<td>~ZF</td>
</tr>
<tr>
<td>js</td>
<td>Label</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>Label</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>Label</td>
<td>jnle</td>
<td>~(SF ∨ OF) &amp; ~ZF</td>
</tr>
<tr>
<td>jge</td>
<td>Label</td>
<td>jnl</td>
<td>~(SF ∨ OF)</td>
</tr>
<tr>
<td>jl</td>
<td>Label</td>
<td>jnge</td>
<td>SF ∨ OF</td>
</tr>
<tr>
<td>jle</td>
<td>Label</td>
<td>jng</td>
<td>(SF ∨ OF) ∨ ZF</td>
</tr>
<tr>
<td>ja</td>
<td>Label</td>
<td>jnbe</td>
<td>~CF &amp; ~ZF</td>
</tr>
<tr>
<td>jae</td>
<td>Label</td>
<td>jnb</td>
<td>~CF</td>
</tr>
<tr>
<td>jb</td>
<td>Label</td>
<td>jnae</td>
<td>CF</td>
</tr>
<tr>
<td>jbe</td>
<td>Label</td>
<td>jna</td>
<td>CF ∨ ZF</td>
</tr>
</tbody>
</table>

**Figure 3.15** The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.
Conditional moves

- Similar to conditional jumps, but they move data if certain condition codes are set
  - Benefit: no branch prediction penalty
    - We'll see how this produces faster code in a few weeks
  - In C code: "\( x = ( <\text{cond}> \ ? \ <\text{tvalue}> : <\text{fvalue}> ) \)"

```assembly
cmpq %rax, %rbx
jg L01
movq %rax, %rcx
jmp L02
L01:
  movq %rbx, %rcx
L02:
```

```assembly
movq %rax, %rcx
cmpq %rax, %rbx
cmovg %rbx, %rcx
```
Conditional moves

- Similar to conditional jumps, but they move data if certain condition codes are set
  - Benefit: no branch prediction penalty
    - We'll see how this produces faster code in a few weeks
  - In C code: "\(x = (\ <\text{cond}>\ ?\ <\text{tvalue}>\ :\ <\text{fvalue}>\)"

```assembly
 cmpq %rax, %rbx
 jg L01
 movq %rax, %rcx
 jmp L02
L01:
   movq %rbx, %rcx
L02:
```

```assembly
 movq %rax, %rcx
 cmpq %rax, %rbx
 cmovg %rbx, %rcx
```
C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret
.L3:
    movl %esi, %eax
    ret
```
Example

C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret

.L3:
    movl %esi, %eax
    ret
```
Example

C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)

min:
  cmpl %esi, %edi
  jge .L3
  movl %edi, %eax
  ret

.L3:
  movl %esi, %eax
  ret
```
Example

C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)
min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret
.L3:
    movl %esi, %eax
    ret
```
Textbook’s “Goto code”

- Compilers translate block-structured code to linear code using conditional jumps
  - We can simulate conditional jumps in C using the goto statement
    - General template: "if (<cond>) goto <label>;;"
    - Syntax for labels is the same in C and assembly (colon notation)

- CS:APP: C “goto code” is code that uses only if/goto and goto
  - No blocks (and therefore no “else” blocks or explicit loops)
  - Not a good idea in general!
    - Famous letter by Dijkstra: "Go To Statement Considered Harmful"
  - However, it is useful for pedagogical purposes (closer to assembly)
Example

C code:

```c
if (x < y) {
    printf("A");
} else {
    printf("B");
}
printf("C");
```

C goto code:

```c
if (x >= y) goto L1;
printf("A");
goto L2;
L1:
    printf("B");
L2:
    printf("C");
```

C code:

```c
while (x < 5) {
    x = x - 1;
}
```

C goto code:

```c
goto L2;
L1:
    x = x - 1;
L2:
    if (x < 5) goto L1;
```
Loops

• Basic idea: jump back to an earlier label
• Three basic forms:
  − Do-while loops
  − Jump-to-middle loops
  − Guarded-do loops
• Note: we’ll use goto code in C first
  − Just to avoid unnecessary complication
  − If you can translate a loop into goto code, it's then much easier to convert to assembly
Loops

do <body-statement>
while (<test-expr>);

while (<test-expr>)
<body-statement>

loop:
<body-statement>
if (<test-expr>)
goto loop;

goto test;
loop:
<body-statement>
test:
if (<test-expr>)
goto loop

if (!<test-expr>)
goto done
loop:
<body-statement>
if (<test-expr>)
goto loop
done:
Loops

**Do-while loop**

```plaintext
do
    <body-statement>
while (<test-expr>);
```

**Jump-to-middle loop**

```plaintext
while (<test-expr>)
    <body-statement>
```

**Guarded-do loop**

```plaintext
if (!<test-expr>)
    goto done
```

```plaintext
if (<test-expr>)
    goto loop
```

```plaintext
<body-statement>
```

```plaintext
if (<test-expr>)
goto loop
```

```plaintext
<body-statement>
```

```plaintext
goto loop
```

```plaintext
done:
```
Loops

\[
\text{for } (<\text{init-exp}>; <\text{test-exp}>; <\text{update-exp}>) \\
<\text{body-statement}>
\]

goto test;
loop: 
<\text{body-statement}>
test: 
if (<\text{test-exp}>)
  goto loop

if (!<\text{test-exp}>)
  goto done
loop:
<\text{body-statement}>
if (<\text{test-exp}>)
  goto loop
done:

Jump-to-middle loop

Guarded-do loop
Loops

for (<init-expr>; <test-expr>; <update-expr>)
<body-statement>

Jump-to-middle loop

Guarded-do loop
T/F: We can **always** translate a program from structured code (with if/then and loops) to linear/goto code.
We can always (and automatically!) translate from structured code to linear/goto code

- This is what a compiler does!
- If you’re interested in learning more about how this works, plan to take CS 432 as your systems elective
• Convert the following C function into x86-64 assembly:

```c
int sum = 0;
int x = 1;
while (x < 10) {
    sum = sum + x;
    x = x + 1;
}
```

Hint: Use jump-to-middle for the while loop
Exercise

- Convert the following C function into x86-64 assembly:

```c
int sum = 0;
int x = 1;
while (x < 10) {
    sum = sum + x;
    x = x + 1;
}
```

```
movl    $0, %eax  # sum = 0
movl    $1, %edx  # x = 1
jmp     test     # goto test

loop:
    addl    %edx, %eax  # sum = sum + x
    addl    $1, %edx    # x = x + 1

test:
    cmpl    $10, %edx  # if (x < 10)
    jl      loop       # goto loop
```

Hint: Use jump-to-middle for the while loop.